UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Lenguaje de dominio específico para la generación de pruebas basadas en máquinas de estados

Sergio Rodrigálvarez Sibón

Tutor: Juan de Lara Jaramillo

Julio 2017

Lenguaje de dominio específico para la generación de pruebas basadas en máquinas de estados

AUTOR: Sergio Rodrigálvarez Sibón
TUTOR: Juan de Lara Jaramillo

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio de 2017

Resumen (castellano)

Muchos elementos software (típicamente un objeto de una clase) pasan por estados, sin embargo, entre las pruebas más comunes y utilizadas no están las que comprueban que las transiciones entre estos estados son correctas. Este documento trata de las pruebas basadas en máquinas de estados, y aunque dicho concepto no es nuevo, no es usual realizar este tipo de pruebas.

Las pruebas basadas en máquinas de estados se caracterizan por ser repetitivas, ya que las distintas rutas o caminos que se pueden realizar sobre una máquina de estados tienen un grado muy alto de solapamiento. El objetivo de este TFG es automatizar la creación de estas pruebas.

Se ha desarrollado para tal objetivo un lenguaje de dominio específico, cuyo nombre es Mech, que permite definir los distintos estados de una máquina de estados así como las transiciones entre ellos. Sin embargo, no basta con definir la máquina, ya que hace falta indicar como vamos a recorrer el grafo que representa la máquina para que las pruebas sean lo más exhaustivas posible.

Básicamente hay dos formas de recorrer este tipo de grafos. Una consiste en recorrer todos los nodos al menos una vez, la cual se conoce como cobertura de nodos. La otra consiste en recorrer todas las transiciones al menos una vez, la cual se conoce como cobertura de aristas. Aunque depende la topografía exacta del grafo, la cobertura de aristas suele ser más exhaustiva que la cobertura de nodos.

Cada transición tiene asociado un método y una serie de comprobaciones, por lo que ejecutar una transición equivale a ejecutar el método y las comprobaciones asociadas. Cada camino obtenido con el criterio de recorrido seleccionado se traduce a un test, en el que se realizan las transiciones y comprobaciones correspondientes, en el orden en que aparecen en el camino.

Palabras clave (castellano)

Java, máquina de estados, grafo, pruebas, lenguaje de dominio específico, generación de código

Abstract (English)

Many software elements (typically an object of a class) has states, however, checking if transition between states are correct isn't amongst the most usual and used testing methodologies. This document is about state machine based testing, which is not a new concept, but not a popular one neither.

State machine based testing is characterized by being repetitive, since the several paths through a state machine are very likely to overlap. This document's objective is to make state machine based tests automatically.

For that purpose, we have developed a domain specific language, whose name is Mech, that allow us to define the states and transitions of a state machine. However, defining the state machine is not enough, since we have to specify how the graph representing the state machine must be traversed, so the testing is as exhaustive as possible.

Basically, there are two ways of traversing the graph. The first one is about traversing each node at least once, known as node coverage. The other one is about traversing each edge at least once, known as edge coverage. Even though it depends on the graph topology, edge coverage is usually more exhaustive than node coverage.

Each transition is associated to a method and a list of asserts, so that executing a transition consists on executing a method and the associated asserts. Each path got by selected coverage criteria is translated into a test, in which methods and assert are executed in the order given by the path.

Keywords (inglés)

Java, state machine, graph, testing, domain specific language, code generation

INDICE DE CONTENIDOS

1 Introducción

El ciclo de vida del desarrollo software consta de una serie de fases, todas ellas necesarias e imprescindibles. Dependiendo de la metodología elegida, hay variaciones, pero, a grandes rasgos, este ciclo consta de las siguientes fase: Análisis, Diseño, Desarrollo (o Programación), Pruebas, Implantación y Mantenimiento [3]. Las Pruebas (o testing) son una de esas fases necesarias e imprescindibles. Dentro de esta fase destacan dos grandes grupos de pruebas: las pruebas unitarias y las pruebas de integración (aunque también son importantes las pruebas de validación, sistema y aceptación). Las primeras se centran en asegurar el correcto funcionamiento de los métodos de los módulos, así como que los mismos cumplan con las especificaciones, todo ello de manera individual e independiente. Los segundos tipos pruebas consisten en asegurar el correcto funcionamiento de los módulos de manera conjunta, es decir, integrados los unos con los otros.

Sin embargo, a la hora de crear las pruebas concretas, los encargados de ello a menudo se enfrentan con la tarea de crear un conjunto de pruebas que sean apropiadas para un determinado software. Hay multitud de libros dedicados a distintas metodologías de pruebas [1]. Es usual, en orientación a objetos, describir el comportamiento de los objetos mediante máquinas de estados [2], por lo que se hace necesario desarrollar herramientas que den soporte a estos métodos.

En este documento vamos a tratar las pruebas basadas en máquinas de estados. Es decir, un elemento software, típicamente una clase en lenguajes orientados a objetos, pasa por estados a la hora de ejecutar funciones o métodos sobre dicho elemento. Concretamente, este documento va a tratar sobre el desarrollo de un lenguaje de dominio específico que genere pruebas basadas en máquinas de estados de manera automática. Las pruebas se van a generar recorriendo los posibles caminos de la máquina de estados. Es más sistemático, ahorra más esfuerzo y es menos propenso a errores definir una máquina de estados y generar automáticamente las pruebas basándose en dicha máquina que escribir manualmente el código de pruebas que modela dicha máquina.

1.1 Motivación

Una vez descrito a grandes rasgos que queremos hacer, vamos a desarrollar el por qué, la motivación. Sin embargo, antes necesitamos un poco de contexto. Una máquina de estados es una abstracción que comprende dos elementos fundamentales: estados y transiciones. Por un lado, los estados son las situaciones en las que un determinado elemento puede estar. Las transiciones, por otro lado, son los eventos que producen un cambio de estado en el elemento; aunque se puede tener una transición de un estado a sí mismo. En nuestro caso, los estados se identifican con las fases por las que pasa un objeto

de una clase, mientras que las transiciones se identifican con los métodos que se ejecutan sobre esos objetos y que hacen cambiar el estado del objeto.

Gráficamente, los estados se representan como grafos dirigidos; los estados se corresponden con los nodos, y las transiciones con las aristas. Una transición siempre va del estado en el que se encontraba el objeto antes de que se produjera el evento hasta el estado en el que se va a encontrar el objeto tras producirse el evento. Es decir, dada una transición, el estado desde el que parte la transición se conoce como predecesor, y el estado en el que termina la transición se conoce como sucesor. En este documento se van a utilizar las palabras nodo y estado como sinónimos, así como las palabras transición y arista.

Es importante tener en cuenta que la máquina de estados tiene que reunir una serie de características. La más importante es que tiene que tener nodos o estados iniciales, esto es, nodos desde que los que partan transiciones pero a los que no lleguen transiciones. Estos nodos iniciales se identifican con la creación de un elemento del módulo software a probar, típicamente la creación de un objeto de una clase. Así mismo, el grafo puede tener nodos finales, esto es, nodos a los que llegan transiciones pero de los que no parten transiciones. Estos nodos se identifican con la destrucción de un elemento del módulo a probar, típicamente un la destrucción de un objeto de una clase. Sin embargo, esta última característica, aunque recomendable, no es necesaria.

Introducido el grafo como representación gráfica de una máquina de estados, viene a la mente inmediatamente el concepto de camino. Un camino es una secuencia [n_1, n_2, ... n_M] de nodos, donde cada par de nodos adyacentes, (n_i, n_{i-1}), $1 <= i < M$, está en el conjunto de artistas. Así mismo, la longitud de un camino es definida como el número de aristas que contiene [1].

Con todo lo introducido hasta el momento, no es difícil discernir que un camino se identifica con un test, que comienza con la creación de un objeto, continúa con la ejecución de los métodos correspondientes a las transiciones del camino, en orden, y termina en un estado final (aunque no necesariamente). Tampoco es difícil darse cuenta que cuando se tienen varios caminos distintos, las probabilidades de solapamiento son muy altas, lo que provoca repetición de código en distintas pruebas. Esto hace, en definitiva, que la programación manual de las pruebas basadas en máquinas de estados sea repetitiva, monótona y propensa a errores.

1.2 Objetivos

El objetivo de este Trabajo de Fin de Grado es, pues, diseñar un lenguaje de dominio específico textual como un plugin para el entorno de desarrollo integrado Eclipse que automatice la creación de las pruebas en el lenguaje orientado a objetos java. Dicho lenguaje permitirá definir un grafo de manera textual, asociando métodos y

comprobaciones a las transiciones. A continuación se definirán criterios de recorrido del grafo, que generará caminos que se identificarán con las pruebas a crear. El código generado utilizará la librería Junit.

Al automatizar la creación de pruebas y definir un lenguaje con un mayor nivel de abstracción, se acelera y facilita la creación de dichas pruebas. De la misma forma, se reducen las posibilidades de errores humanos gracias a la dicha automatización.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- Introducción: Este capítulo es en el que estamos ahora. Se desarrolla la motivación de este trabajo, los objetivos que se esperan alcanzar y la distribución de la memoria

- Estado del arte: En este capítulo se va a hablar de una serie de conceptos previos y herramientas con los que se va a realizar el trabajo expuesto. También se va a exponer el trabajo relacionado con el campo de conocimiento del proyecto.

- Diseño: Se va a exponer de manera abstracta la estructura del sistema que se va a desarrollar, con el objetivo de mostrar claramente cuáles son las partes de dicho sistema.

- Desarrollo: En este capítulo se va a mostrar se manera mucha más concreta que en el capítulo anterior cómo y con qué herramientas se va a realizar el sistema objetivo.

- Integración, pruebas y resultados: Vamos a desarrollar un ejemplo de funcionamiento del sistema con el objetivo de mostrar la corrección del mismo.

- Conclusiones y trabajo futuro: En este capítulo se va a hablar de las conclusiones y las reflexiones tras desarrollar el proyecto. De la misma forma se van a hablar de las posibilidades de trabajo futuro relacionadas con el campo de conocimiento del proyecto que se ha desarrollado.

2 Estado del arte

2.1 Conceptos previos

Para la realización del trabajo que tenemos entre manos, vamos a disponer de una serie de herramientas y conceptos que nos van a facilitar la tarea: el diseño de un lenguaje de dominio específico para la realización de pruebas basadas en máquinas de estados. Estas herramientas y conceptos son los siguientes:

- **Máquinas de estados UML**: Diagrama de comportamiento que modela una máquina finita de estados dentro del lenguaje de modelado unificado [7].
- **Junit**: Librería java para la realización de pruebas unitarias [8].
- **Xtext**: Marco de trabajo java para desarrollo de lenguajes de dominio específico [9].
- **Xtend**: Lenguaje que se traduce a código java y que extiende la funcionalidad de este último [10].
- **EMF**: Marco de trabajo de modelado y generación de código a partir de modelos [11].

2.1.1 Máquinas de estados UML

Una máquina de estados UML se utiliza para especificar un comportamiento discreto de un artefacto software [7]. Hablamos de comportamiento discreto porque tenemos una serie finita de estados por los que puede pasar el artefacto software. Las máquinas de estados UML se modelan visualmente como un grafo dirigido en que los vértices se identifican con los estados y las aristas que unen los vértices se identifican con las transiciones. Una transición siempre va de un estado inicial a un estado destino, aunque el estado inicial y el estado destino pueden ser el mismo. En la figura 2.1. tenemos un ejemplo de máquina de estados UML.

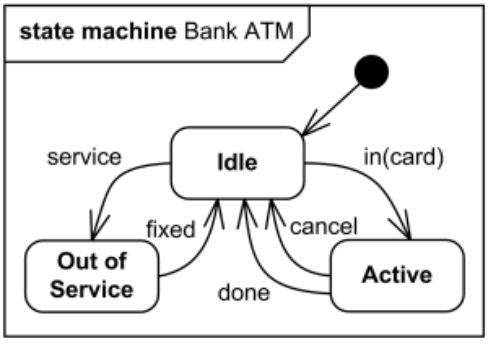

Figura 2.1. Ejemplo de Máquina de Estados UML

En la figura 2.2 tenemos un ejemplo de estado compuesto UML. Se trata de un estado que tiene subestados anidados. Sin embargo, a esta característica no se le va a dar soporte en nuestro lenguaje de dominio específico.

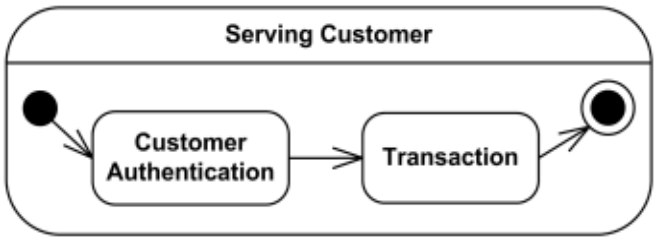

Figura 2.2. Ejemplo de estado compuesto UML

El punto negro de la izquierda en la figura 2.2. corresponde con un punto de entrada. Este punto de entrada indica cual es el estado incial en el que comienza la máquina de estados (en este ejemplo, "Customer Authentication"). Mientras que el punto negro rodeado por un círculo blanco en la derecha de la figura 2.2. indica un punto de salida. Un punto de salida índica un estado en el que termina la máquina de estados (en este ejemplo, "Transaction"). Algunas características adicionales de una máquina de estados son [7]:

- **Elección:** Punto en el que se evalúa una condición según la cual se elige una transición u otra, como se observa en la figura 2.3. Se puede emular con nuestro lenguaje mediante dos transiciones, cada una con una guarda.

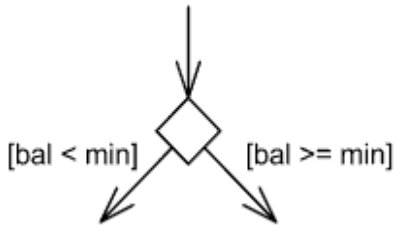

Figura 2.3. Ejemplo de elección

- **Bifurcación:** Punto en el que una transición se divide en dos o más transiciones, como se observa en la figura 2.4. No se le va a dar soporte a esta característica porque se utiliza en sistemas concurrentes.

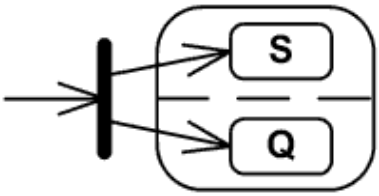

Figura 2.4. Ejemplo de bifurcación

▪ **Join:** Comportamiento complementario al anterior. Dos o más transiciones convergen en una, como se observa en la figura 2.5. No se le va a dar soporte a esta característica porque se utiliza en sistemas concurrentes.

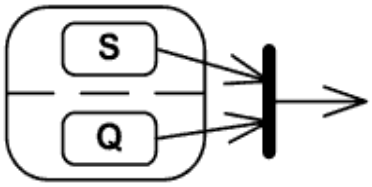

Figura 2.5. Ejemplo de convergencia UML

▪ **Guardas:** Condición que se ha de cumplir para que una transición tenga lugar.

También pueden existir estados jerárquicos o regiones concurrentes [5]. Sin embargo, no se va a dar soporte a estas características tampoco.

2.1.2 Junit

Junit es una librería muy conocida para la realización de pruebas unitarias, es decir, pruebas de métodos de manera aislada. Estas pruebas consisten fundamentalmente en comparar un resultado, obtenido con nuestro software, con un valor esperado.

Una vez importadas las librerías necesarias, tendremos acceso a una serie de métodos de comprobación sobrecargados para tipos primitivos, objetos y arrays [12][13]. Existen más métodos en la librería de Junit, pero estos son los que nos interesan:

▪ assertArrayEquals: Comprueba que dos arrays son iguales.

▪ assertEquals: Si los dos elementos a comparar son de tipo primitivo, comprueba que son iguales. Si los dos elementos a comparar son objetos, comprueba que son iguales con el método "equals" de la clase.

▪ assertTrue: Comprueba que una condición booleana es cierta.

▪ assertFalse: Comprueba que una condición booleana es falsa.

▪ assertNotNull: Comprueba que un objeto no es null.

▪ assertSame: Comprueba que dos referencias apuntan al mismo objeto.

▪ assertNotSame: Comprueba que dos referencias no apuntan al mismo objeto.

Un test Junit simple empieza con la declaración de una clase. Dentro de esta clase, vamos a definir métodos con la etiqueta "@Test", dentro de los cuales vamos a realizar nuestras comprobaciones con los métodos "assert", tal y como se ve en la figura 2.6.

```
public class SimpleTest {

        @Test
        public void testEmptyCollection () {
          Collection collection = new ArrayList ();
          assertTrue (collection.isEmpty());
        }
}
```

Figura 2.6. Ejemplo simple de Test Junit

Dada la integración de Junit con la mayoría de IDE, no suele hacer falta definir un método main. El IDE suele mostrar de manera gráfica los resultados del test.

2.1.3 Xtext

Xtext es un marco de trabajo para el desarrollo de lenguajes de dominio específico con sintaxis textual. Nos provee con una serie de herramientas típicas en el campo de los compiladores, tales como analizador morfológico y analizador sintáctico. Basta con definir el lenguaje en forma de gramática libre de contexto. Xtext generará un meta-modelo a partir del lenguaje definido, aunque también se puede partir de un meta-modelo ya construido.

Cada vez que guardemos el archivo en el que estamos escribiendo código en nuestro DSL, se ejecutará el método de generación de código, en que tendremos acceso a una instancia del meta-modelo. Se tratará más en detalle en la sección de Desarrollo.

2.1.4 Xtend

Xtend es un lenguaje que extiende el lenguaje java, dándole funcionalidad extra, y que se traduce a código java, de la misma forma que un compilador traduce código C a código máquina. Algunas de las funcionalidades destacables son:

- **Inferencia de tipos:** El propio compilador del lenguaje es capaz de inferir el tipo de una variable si tiene suficiente información. Es una característica de los lenguajes funcionales.

- **Expresiones lambda:** También conocidas como funciones anónimas. Son funciones que se definen sin nombre, indicando solo argumentos de entrada, retorno y código. Son especialmente útiles cuando se trata de funciones breves que se van a usar solo una vez. Aunque Java lo soporta desde su versión 8, Xtend lo soporta desde hace más tiempo.

- **Expresiones plantilla:** Son métodos que devuelven una concatenación legible de cadenas de texto tabuladas [6]. Se tratará más en detalle en la sección de Desarrollo.

- **Sobrecarga de operadores:** Permite dar nueva funcionalidad a los operadores usuales en los lenguajes de programación.

Xtend, gracias a las expresiones plantilla, nos va a facilitar la tarea de generar código java junit a partir de nuestro DSL.

2.1.5 EMF

EMF es un marco de trabajo de modelado y generación de código a partir de modelos. El meta-modelo generado por Xtext a partir del DSL es realizado con las herramientas de este marco de trabajo. Los meta-modelos de EMF son almacenado en archivos ecore.

Cabe destacar también el editor de diagramas Ecore Tools, que permite ver y generar meta-modelos ecore de manera visual [14].

2.2 Trabajo relacionado

La idea de realizar pruebas basadas en máquinas de estados no es nueva [15][16]. Sin embargo, no tenemos constancia de que haya un procedimiento estandarizado ni herramientas que faciliten y automaticen la creación de este tipo de pruebas. De esta carencia partió la idea de crear un lenguaje de dominio específico para realizar esta tarea.

Dado que convertir un objeto con estados a una máquina de estados UML es un procedimiento trivial, la complejidad del trabajo estaba en cómo generar las pruebas a partir de dicha máquina. La idea estribaba en elegir un estado e ir aplicando transiciones, comprobando que se realizaban correctamente los cambios de estado. Sin embargo nos faltaba un criterio claro de cómo realizar esta tarea. En "Coverage Criteria for State Based Specifications" [1] se proponen dos criterios principales:

- **Cobertura de nodos**: Consiste en generar caminos desde un estado inicial hasta un estado final de la máquina de estados, hasta que todos los estados se hayan visitado al menos una vez.

- **Cobertura de aristas**: Consiste en generar caminos desde un estado inicial hasta un estado final de la máquina de estados, hasta que todas las transiciones se hayan visitado al menos una vez.

Estos criterios no tienen en cuenta la no existencia de estados finales o la existencia de bucles, pero en la sección de Diseño trataremos estos problemas en detalle.

3 Diseño

Como ya se ha expuesto en los capítulos anteriores, vamos a desarrollar un lenguaje de dominio específico que permita la generación automática de pruebas basadas en máquinas de estados. Dichas pruebas se van a generar recorriendo los posibles caminos de una máquina de estados siguiendo distintos criterios. La automatización de este proceso es el principal objetivo. El nombre del lenguaje va a ser **Mech**.

En esta sección vamos a tratar el diseño del lenguaje, así como sus componentes a alto nivel, entrando en la implementación concreta del mismo en la siguiente sección.

3.1 Arquitectura

Como todo lenguaje de modelado **Mech**, consta de tres componentes principales [4]:

- **Sintaxis abstracta**, definida por un meta-modelo
- **Sintaxis concreta**, textual en este caso
- **Semántica**, relacionada con la generación de código

En las siguientes secciones se tratará en detalle cada una de estos componentes en detalle, entrando en los aspectos concretos de los mismos.

3.2 Sintaxis abstracta

Para definir la sintaxis abstracta de **Mech**, se ha definido un **meta-modelo** que representa los distintos componentes del lenguaje y la relación entre ellos, tal y como se observa en la figura 3.1. Nuestro meta-modelo se compone de los siguientes elementos, que se irán detallando en la siguientes subsecciones, de ser necesario:

- **Domainmodel**: Es la clase raíz que contiene todas las demás.
- **AbstractState**: Es la superclase de la que hereden InitialState y State.
- **InitialState**: Esta clase representa los estados iniciales de la máquina de estados, es decir, los estados de los que parten transiciones y a los que no llegan transiciones.
- **State**: Esta clase representa el resto de estados de la máquina de estados, incluidos los estados finales, de haberlos.
- **Transition**: La clase que modela los métodos que provocan cambio de estado en el objeto.

▪ **Command**: La clase que modela los métodos asociados a una transición.

▪ **Assert**: Esta clase representa las comprobaciones que se van a realizar tras ejecutar el método que provoca el cambio de estado.

▪ **Imports**: Necesario para importar la clase que se va a probar, así como otras clases que se vayan a utilizar.

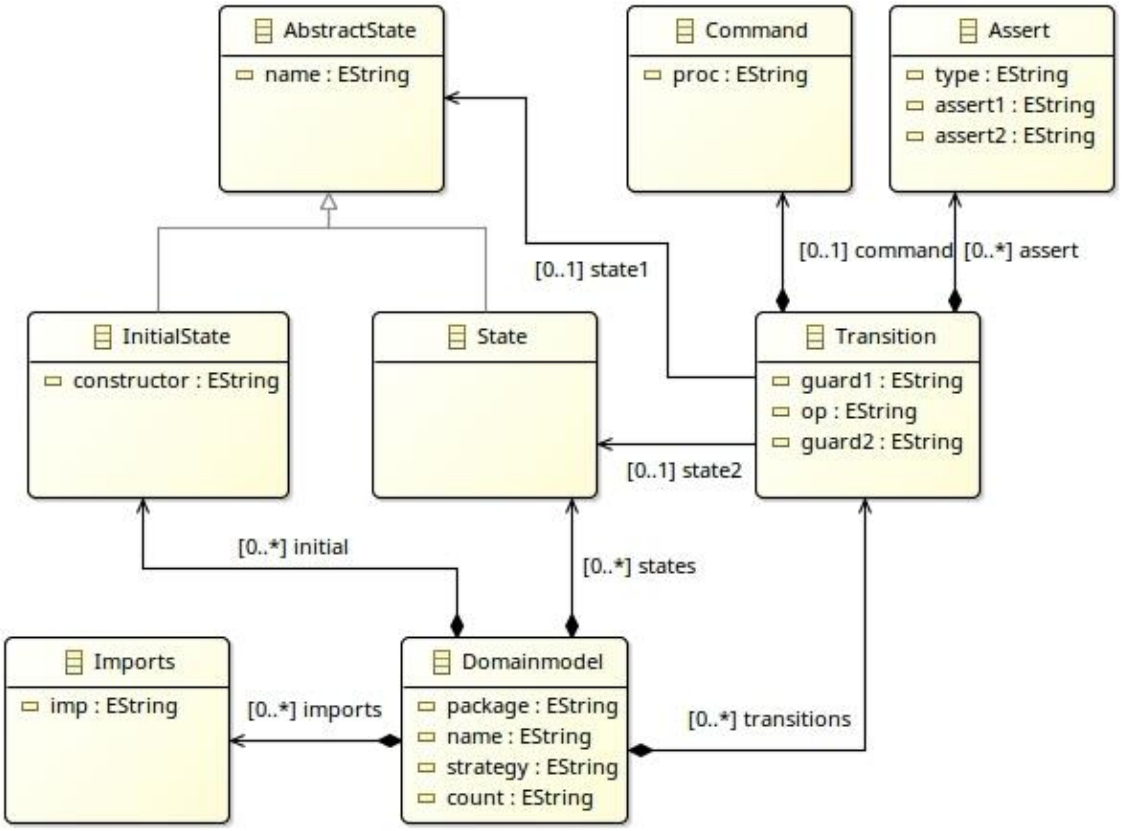

Figura 3.1. Meta-Modelo del lenguaje de dominio específico Mech

3.2.1 Domainmodel

Esta es la clase raíz que contiene todas las demás, así como la que marca la estrategia de recorrido la máquina de estados y la política de bucles. La clase Domainmodel consta de los siguientes atributos:

▪ **Name**: Nombre de la clase sobre la que se va a ejecutar las pruebas basadas en máquina de estados.

▪ **Strategy**: Estrategia de recorrido de la máquina de estados. Los caminos se generan de manera aleatoria siguiendo una estrategia. Un camino termina una vez que se ha alcanzado un estado final, o un bucle. Existen fundamentalmente tres estrategias de recorrido de máquina de estados en nuestro lenguaje [1]:

- **state**: Esta estrategia se denomina cobertura de estados [1] y se basa en pasar por cada estado al menos una vez. Vamos generando caminos aleatorios, y una vez que hemos pasado por cada estado al menos una vez, paramos la generación. Si el grafo de la máquina de estados es denso (hay más aristas que vértices), existen altas probabilidades de que el conjunto de caminos generado no incluya todo las transiciones presentes en el grafo, por lo que esta estrategia es menos exhaustiva que la siguiente que vamos a tratar.

- **transition**: Esta estrategia se denomina cobertura de transiciones [1] y se basa en pasar por cada transición al menos una vez. Vamos generando caminos aleatorios, y una vez que hemos pasado por cada transición al menos una vez, paramos la generación. Como se ha explicado en la estrategia anterior, esta estrategia es más exhaustiva que la anterior.

- **both**: Esta estrategia ejecuta ambas estrategias y genera tests para ambas estrategias.

- **LoopCount**: Como hemos desarrollado anteriormente, un camino termina una vez que se encuentra un estado final o un bucle. Si termina con un bucle, podemos querer recorrer varias veces el bucle antes de terminar; este atributo nos permite determinar cuántas veces queremos recorrer dicho bucle con un número entero.

- **Package**: El paquete en el que se va a localizar el código generado.

- **Imports**: Clases que vamos a necesitar para la ejecución de las pruebas.

- **Initial**: Lista de estados iniciales. Se desarrollará más adelante.

- **States**: Lista del resto de estados. Se desarrollará más adelante.

- **Transitions**: Lista de transiciones entre estados. Se desarrollará en un subsección.

3.2.2 AbstractState

Esta clase es la superclase de la que heredan InitialState y State. Es necesaria, ya que a la hora de manipular estados, a veces nos da igual si son iniciales o no. El único atributo de esta clase es el nombre del estado.

3.2.3 InitialState

Esta clase representa los estados iniciales, es decir, los estados de los que parten transiciones pero a los que no llegan transiciones. Estos estados son el punto de comienzo de cualquier camino y suponen la creación de un objeto de la clase que se está probando con **Mech**. Tiene un atributo opcional, que es el constructor con el que se va a crear el objeto; de no especificar este atributo, se utilizará el constructor por defecto.

3.2.4 State

Esta clase representa el resto de estados, es decir, aquellos que no son iniciales. No tienen ningún atributo, salvo el heredado de AbstractState, pero son necesarios para representar los estados por los que pasa un determinado objeto.

3.2.5 Transition

Esta clase representan las transiciones que producen cambios de estado. Las transiciones pueden tener asociada una guarda; si dicha guarda no se cumple, no se ejecuta el método asociado a la transición y se termina la ejecución del test asociado al camino. Tiene asociados los siguientes atributos:

- **Command**: Método de la clase a probar que provoca el cambio de estado del objeto.
- **Assert**: Lista de comprobaciones Junit que se van a ejecutar tras la ejecución del método.
- **State1**: Estado del que parte la transición.
- **State2**: Estado al que llega la transición.
- **Guard1:** Elemento a la izquierda de la comparación.
- **Guard2:** Elemento a la derecha de la comparación.
- **Op:** Operador de comparación seleccionado. Puede ser cualquiera de los comparadores comunes.

3.2.6 Assert

Esta clase representa un assert Junit. Tiene asociados dos atributos, que son los elementos a comparar:

- **Assert1:** Primer argumento de la comprobación.
- **Assert2:** Segundo argumento de la comprobación.
- **Type:** Tipo de comprobación Junit. Puede ser AssertEquals, AssertSame o AssertNotSame.

3.3 Sintaxis concreta

Mech utiliza una sintaxis concreta textual. Esto es debido principalmente a la naturaleza de las herramientas que se han utilizado para desarrollar el DSL; también influye el hecho de que el usuario objetivo va a ser un programador, que está acostumbrado a usar lenguajes de programación, que utilizan sintaxis textual, además de que se va a integrar con java.

La sintaxis concreta se va a exponer en las siguientes subsecciones, desglosándola en sus distintas componentes, de forma similar a como se ha hecho en la sección anterior. La nomenclatura que se va a seguir es la siguiente: los elementos fijos se encuentra entrecomillados, mientras que los elementos variables (clases, identificadores y enteros) se encuentran rodeados por los siguientes signos <>. Por lo demás, se sigue nomenclatura de expresiones regulares. La sintaxis completa está en el Anexo A.

3.3.1 Domainmodel

La sintaxis de la clase root está expuesta en la figura 3.2. Los atributos de esta clase se describieron en detalle en la sección anterior (Sintaxis Abstracta), sin embargo, vamos a recordarlos en su orden de aparición:

- **Package**: Este atributo indica el paquete en el que se va a situar la prueba basada en máquina de estados, una vez generado.

- **Imports**: Lista de imports de clases necesarias para la ejecución de la prueba.

- **Name**: Nombre de la clase java a probar.

- **Coverage Strategy**: Estrategias de cobertura. Se desarrollaron en detalle en la sección anterior.

- **LoopCount**: Número de veces que se va a recorrer un bucle antes de finalizar el camino.

- **InitialState**: Lista de estados iniciales. Cualquier prueba tiene que tener al menos un estado inicial, desde los que partirán los caminos generados.

- **State**: Lista del resto de estados. Puede haber cero o más estados.

- **Transition**: Lista de transiciones entre los distintos estados. Puede haber cero o más transiciones.

```
“package” <Import>
<Imports>*

“spec for” <ID> “coverage” (“state” | “transition” | “both”) “loopCount” <INT> “{”
    <InitialState>+
    <State>*
    <Transition>*
“}”
```

Figura 3.2. Sintaxis concreta de la clase root

3.3.2 ID

La sintaxis del elemento terminal ID está expuesta en la figura 3.3. Este elemento terminal representa una cadena de texto para identificadores únicos java (clases, objetos, etc). Son elementos terminales.

```
“^”? (“a” .. “z” | “A” .. “Z” | “_”) (“a” .. “z” | “A” .. “Z” | “_” | “0” .. “9”)*
```

Figura 3.3. Sintaxis concreta de la clase ID

3.3.3 INT

La sintaxis del elemento terminal INT está expuesta en la figura 3.4. Este elemento terminal representa números enteros.

```
(“0” .. “9”)+
```

Figura 3.4. Sintaxis concreta de la clase INT

3.3.4 Import

La sintaxis del elemento terminal Impot está expuesta en la figura 3.5. Este elemento terminal representa la notación de punto java, típica en lenguajes orientados a objetos.

```
(<ID> “.”)* <ID>
```

Figura 3.5. Sintaxis concreta de la clase Import

3.3.5 Imports

La sintaxis de la clase Imports está expuesta en la figura 3.6. Esta clase representa un import tipo java, que se escribirá tal cual en el código generado.

```
"import" <Import>
```

Figura 3.6. Sintaxis concreta de la clase Imports

3.3.6 InitialState

La sintaxis de la clase InitialState está expuesta en la figura 3.7. Esta clase representa un estado inicial. Como ya se expuso en la sección anterior, un estado inicial supone la creación de un objeto de la clase que se está probando. Se puede indicar el constructor a utilizar si se desea, de lo contrario, se utiliza el constructor por defecto.

```
"initial" "state" <ID> ("{"
        <Method>
"}")?
```

Figura 3.7. Sintaxis concreta de la clase InitialState

3.3.7 State

La sintaxis de la clase State está expuesta en la figura 3.8. Esta clase representa el resto de estados.

```
"state" <ID>
```

Figura 3.8. Sintaxis concreta de la clase State

3.3.8 AbstractState

La sintaxis de la clase AbstractState está expuesta en la figura 3.9. Esta clase representa la superclase de la que heredan los estados iniciales y el resto de estados.

```
<State> | <InitialState>
```

Figura 3.9. Sintaxis concreta de la clase AbstractState

3.3.9 Transition

La sintaxis de la clase Transition está expuesta en la figura 3.10. Esta clase tiene los siguientes atributos en orden de aparición:

- **Estado origen**: estado de la clase AbstractState del que parte la transición. Es el primer identificador.
- **Estado destino**: estado de la clase State al que llega la transición. Es el segundo identificador.
- **Guarda**: comparación opcional que determina si se va a ejecutar el método asociado a la transición y si se va a continuar con el test asociado al camino.
- **Command**: método a ejecutar asociado a la transición.
- **Assert**: Lista de comprobaciones a realizar tras ejecutar el método asociado a la transición

```
“transition” <ID> “-” <ID> “}”
    (“guard” (<Method> | <NUM>) (“==” | “!=” | “>” | “>=” | “<” | “<=”) (<Method> | <NUM>))?
    <Command>
    <Assert>*
“}”
```

Figura 3.10. Sintaxis concreta de la clase Transition

3.3.10 Command

La sintaxis de la clase Command está expuesta en la figura 3.11. Esta clase representa una llamada a un método asociado a una transición.

```
“proc” Proc
```

Figura 3.11. Sintaxis concreta de la clase Proc

3.3.11 Assert

La sintaxis de la clase Assert está expuesta en la figura 3.12. Cada assert o comprobación tiene dos argumentos, que pueden ser un método o un número. Dichos argumentos son los elementos a comparar en el Assert, el cual se traduce a un assert Junit.

```
(“assertEquals” | “assertSame” | “asssertNotSame”) (<Proc> | <NUM>) “,” (<Proc> | <NUM>)
```

Figura 3.12. Sintaxis concreta de la clase Assert

3.3.12 Proc

La sintaxis del elemento terminal Proc está expuesta en la figura 3.13. Este elemento

```
(<ID> | <Method>) (“.” (<ID> | <Method>) )*
```

terminal representa una llamada a un método o en un atributo en notación de punto java.

Figura 3.13. Sintaxis concreta de la clase Proc

3.3.13 Method

La sintaxis del elemento terminal Method está expuesta en la figura 3.14. Este elemento terminal representa un método en notación java, con sus argumentos.

```
<ID> “(” ( ( <Proc> | <NUM> ) ( “,” ( <Proc> | <NUM> ) )* )? “)”
```

Figura 3.14. Sintaxis concreta de la clase Method

3.4 Generador de código

La parte de generación de código corresponde con la semántica del lenguaje. A partir de código DSL de Mech, vamos a generar código java Junit, de la misma forma, que, por ejemplo, el lenguaje C es traducido a código máquina por el compilador. El generador de código realiza las siguientes acciones:

- El "package" y los "imports" son incrustados tal cual en el código.
- Se genera un grafo que representa la máquina de estados que se está modelando.
- Se ejecutan la estrategia y el "loop count" elegidos sobre el grafo, lo cual devuelve una lista de caminos.
- Se crea una clase java en la que cada camino es un test Junit.
- Cada test Junit comienza con la creación de un objeto, utilizando el constructor especificado, o el constructor por defecto en caso de no haberse especificado ninguno.
- Tras la creación del objeto, se recorre el camino asociado al test, y por cada transición en el camino, se incrusta la guarda, si se ha especificado, el método y las comprobaciones asociadas a dicha transición.

4 Desarrollo

El código se encuentra en: https://github.com/SeRodrigalvarez/Mech.git

En este capítulo se va a mostrar los detalles de desarrollo e implementación del diseño expuesto en el capítulo 3. Como en el capítulo anterior, vamos a diferenciar tres partes del lenguaje de dominio específico que estamos desarrollando:

- La **sintaxis abstracta** está definida por un meta-modelo, que está generado a partir de la sintaxis textual por el marco de trabajo **Xtext**, por lo que no hace falta que la diseñemos manualmente. El meta-modelo mostrado en la sección de Diseño es el generado por Xtext.

- La **sintaxis textual concreta** está escrita con las herramientas provistas por el marco de trabajo **Xtext**, que hace sencillo el diseño del lenguaje.

- La **generación de código** se realiza con el lenguaje **Xtend**, que está integrado por defecto con el Xtext.

4.1 Sintaxis abstracta

Como acabamos de exponer, la sintaxis abstracta está definida por un meta-modelo generado automáticamente por el marco de trabajo **Xtext**. A partir de la definición de la sintaxis textual, Xtext infiere las clases y nos genera un archivo Ecore que modela el meta-modelo, como se muestra en la figura 4.1.

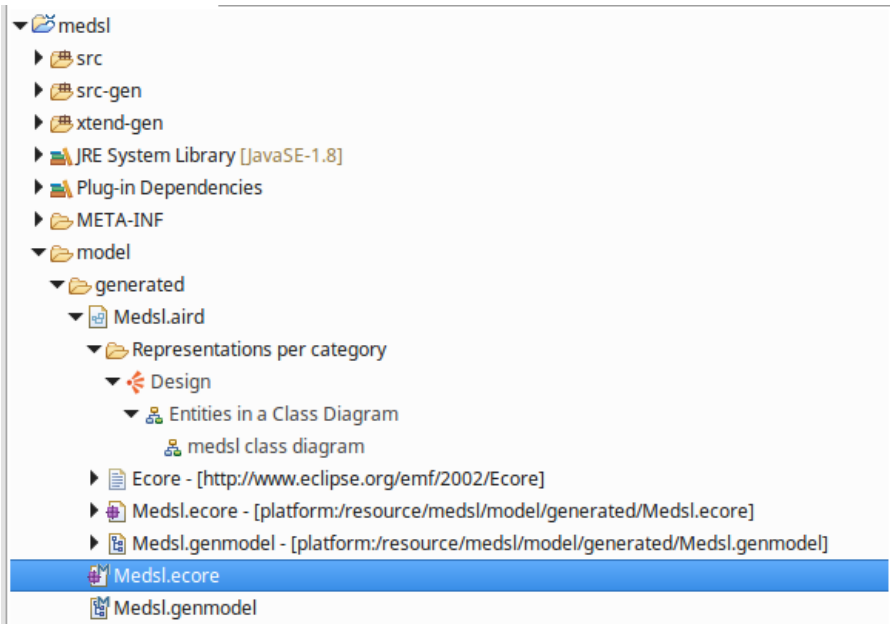

Figura 4.1. Archivo Ecore generado por Xtext y representación gráfica con Ecore Tools

Sin embargo, el archivo Ecore se muestra por defecto como una serie de desplegables tal y como se muestra en la figura 4.2. Para obtener una representación gráfica, necesitaremos el marco de trabajo **Ecore Tools**. Con dicho marco de trabajo podemos generar varias representaciones gráficas del meta-modelo mediante la creación de un archivo **aird**. En nuestro caso, hemos decidido representarlo como un diagrama de clases, tal y como se observa en la figura 4.2. y en la figura 3.1. del capítulo anterior.

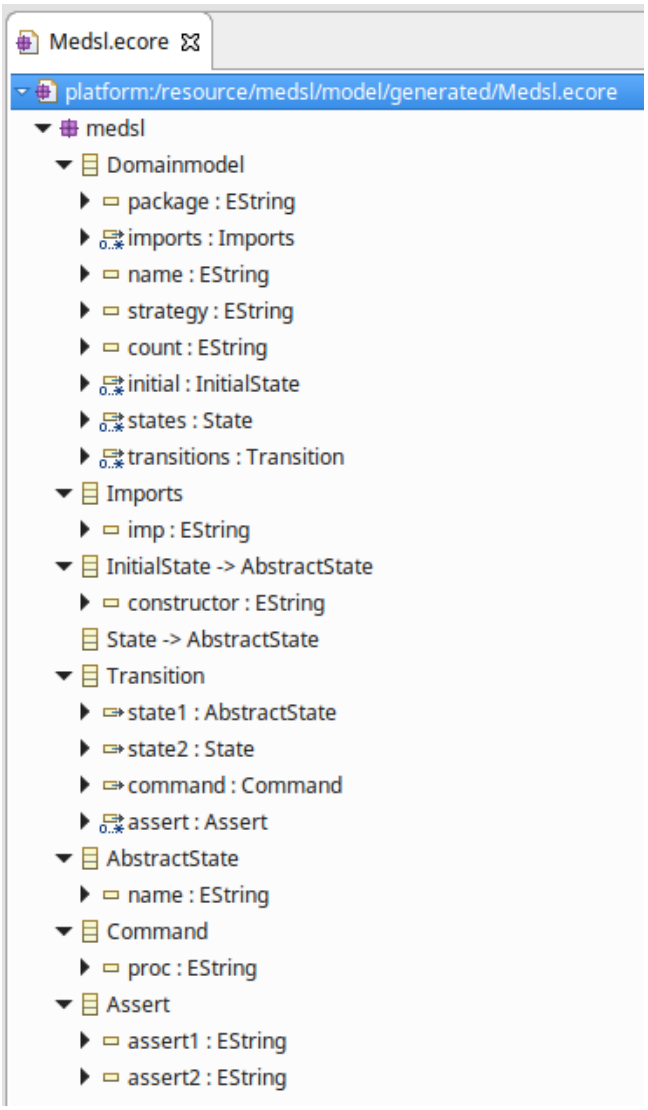

Figura 4.2. Representación por defecto del archivo ecore

4.2 Sintaxis concreta

Como ya se ha expuesto anteriormente, vamos a utilizar el marco de trabajo **Xtext** para definir nuestro lenguaje de domino especifico. Veremos que la sintaxis concreta textual con Xtext guarda similitudes con la notación que hemos usado en el capítulo anterior; esto es debido a que es notación usual en el desarrollo de compiladores. Al fin y al cabo, lo que estamos desarrollando es un compilador con las herramientas de las que nos provee el marco de trabajo.

Dado que ya hemos desarrollado en detalle la sintaxis textual en el capítulo anterior, comentando las diferentes clases y elementos terminales de manera individual, en esta sección vamos a mostrar la implementación con Xtext en partes y la vamos a comentar. Nuestra definición se escribe en un archivo de extensión xtext, como se muestra en la figura 4.3.

Medsl.xtext

```
1  grammar es.uam.eps.Medsl with org.eclipse.xtext.common.Terminals
2
3  generate medsl "http://www.uam.es/eps/Medsl"
4
5  Domainmodel:
6
7      'package' package=Import
8
9      imports+=Imports*
10
11     'spec' 'for' name=ID
12         'coverage' strategy=('state' | 'transition' | 'both')
13         'loopCount' count=NUM '{'
14         initial+=InitialState+
15         states+=State*
16         transitions+=Transition*
17     '}';
```

Figura 4.3. Parte 1 de la definición de la sintaxis textual con Xtext

Con la primera línea, declaramos el nombre de nuestra gramática e incluimos los terminales por defecto de Xtext (solo vamos a usar ID). Nuestra primera definición es la de la clase root Domainmodel. La variable "package" es un ejemplo de atributo al que se le asigna un objeto de una clase con el operador "=", en este caso, un objeto de la clase Import. Mientras que la variable imports es un ejemplo de atributo lista, al que se le puede asignar varios objetos de una misma clase, en este caso, objetos de la clase Imports. Como en notación de expresiones regulares, el operador "*" indica que puede haber cero o más elementos, mientras que el operador "+" indica que puede haber uno o más elementos.

```
Imports:
    'import' imp=Import
;

InitialState:
    'initial' 'state' name=ID ('{'
        constructor=Method
    '}')?
;

State:
    'state' name=ID;

Transition:
    'transition' state1=[AbstractState] '-' state2=[State]
    ('guard' guard1=(Method | NUM) op=('==' | '!=' | '>' | '>=' | '<' | '<=') guard2=(Method | NUM))?
    '{'
        command=Command
        assert+=Assert*
    '}';

AbstractState: State | InitialState;
```

Figura 4.4. Parte 2 de la definición de la sintaxis textual de Xtext

En la figura 4.4. lo más interesante son los atributos state1 y state2 de la clase Transition, ya que son referencias cruzadas; es decir, se comprueba que se haya declarado previamente los estados que se referencian ahí. Lo cual nos ahorra el trabajo de validación relacionado con transiciones entre estados que no existen. Merece la pena mencionar que la clase AbstractState se creó para indicar que una transición puede partir de cualquier estado, mientras que solo puede terminar un estado que no sea inicial.

```
Command: 'proc' proc=Proc;

Assert: type=('assertEquals' | 'assertSame' | 'asssertNotSame') assert1=(Proc | NUM) ',' assert2=(Proc | NUM);

Proc: (ID | Method)('.'(ID | Method))*;

Method: ID ('<' ID '>')? '(' ((Proc | NUM) (',' (Proc | NUM))*)? ')';

//Assign: Proc Proc? '=' ('new')? Proc;

Import: (ID '.')*ID;

terminal NUM: ('0'..'9')+;
```

Figura 4.5. Parte 3 de la definición de la sintaxis textual de Xtext

En la figura 4.5. tenemos el terminal NUM, que sobrescribe el terminal NUM de Xtext. La razón de esto es debida a que el terminal NUM de Xtext devuelve un entero, mientras que a nosotros necesitábamos que devolviera una cadena de texto que representara el número, para que no hubiera problemas con los tipos de los atributos de la clase Assert.

4.3 Generación de código

La generación de código se realiza con **Xtend**. Xtend, como ya se ha desarrollado en la sección de 2, es un dialecto de Java que añade funcionalidades que este último no presenta, como inferencia de tipos, sobrecarga de operadores, etc. Un archivo fuente escrito en Xtend es traducido a Java. Entre las funcionalidades de Xtend, existe una que lo hace ideal para el trabajo de generación de código: las "template expressions" (en español se traduciría como "expresiones plantilla").

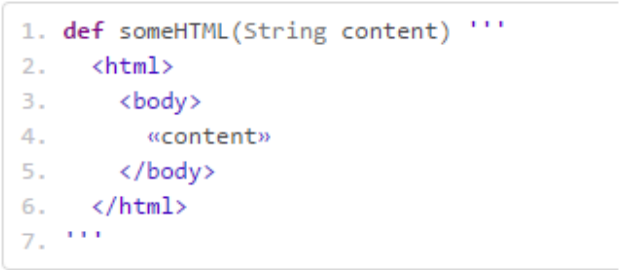

Figura 4.6. Ejemplo de "template expression"

Una "template expression" es un método que se abre y cierra con triples comillas (como se observa en la figura 4.6.) y que devuelve una concatenación legible de cadenas de texto tabuladas [6]. Esta funcionalidad permite crear un fragmento de texto en el que la tabulación importa, de manera fácil y legible. Esto va a facilitar la creación de código java correctamente tabulado, que es a lo que vamos a traducir nuestro lenguaje. Los signos « » se utilizan para intercalar expresiones (llamadas a funciones, acceso a variables, etc).

En la figura 4.7. tenemos la primera parte de la clase encargada de la generación de código. Tenemos funciones para gestionar un contador, necesario para numerar las distintas pruebas que vamos a generar. La función "doGenerate" se llama automáticamente cada vez que hay una modificación en el código fuente de nuestro lenguaje de dominio específico. En este método vamos a crear el archivo java en el que vamos a escribir el código junit y vamos a indicar con que método iniciar la generación.

```
25 class MedslGenerator extends AbstractGenerator {
26
27     var int counter;
28     var Domainmodel dm;
29
30     def void iniCounter() {
31         counter=0;
32     }
33
34     def void incCounter() {
35         counter++;
36     }
37
38     def int getCounter() {
39         return counter;
40     }
41
42     @Inject extension IQualifiedNameProvider
43     override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {
44
45         this.dm = resource.allContents.findFirst[EObject e | e instanceof Domainmodel] as Domainmodel;
46
47         fsa.generateFile(
48                 dm.package.replace('.','/')+"/"+ dm.fullyQualifiedName + "Test.java",
49                 compile)
50
51     }
```

Figura 4.7. Parte 1 de la generación de código

En la figura 4.8. tenemos el método "compile", que es con el que vamos a comenzar la construcción de la prueba, y el método "generateTests". Este último va a crear un grafo con la información contenida en la instancia del meta-modelo y va a obtener la estrategia de recorrido de grafos y la cantidad de veces que se ha de iterar un bucle. Si se ha elegido el recorrido de estados, se llamará a "generateNodeCoverageTests", si se ha elegido el recorrido de transiciones, se llamará a "generateEdgeCoverageTests", y si se ha elegido ambas, se llamará a ambos métodos. Vamos a explicar las dos estrategias (El pseudocódigo de los algoritmos se encuentra en el anexo B):

- **Cobertura de nodo:** En un principio, todos los nodos están en la lista de nodos no visitados. Cogemos un nodo aleatorio de la lista, aplicamos una búsqueda en anchura hasta dicho nodo y generamos un camino aleatorio desde el mismo; la concatenación de ambos resultados nos da un camino. En el proceso vamos eliminando de la lista aquellos nodos por los que hemos pasado. Este proceso se repite hasta que no quedan nodos en la lista.

- **Cobertura de aristas:** En un principio, todas las aristas están en la lista de aristas no visitadas. Cogemos una arista al azar de la lista, aplicamos una búsqueda en anchura hasta el nodo origen de la transición, y generamos un camino aleatorio desde el nodo destino de la transición; la concatenación de los dos resultados con la transición seleccionada nos da un camino. En el proceso vamos eliminando aquellas transiciones por las que hemos pasado. Este proceso se repite hasta que no quedan transiciones en la lista.

```
53      def compile() '''
54          package «dm.package»;
55
56          «FOR i:dm.imports»
57              import «i.imp»;
58          «ENDFOR»
59
60          import static org.junit.Assert.assertEquals;
61
62          import org.junit.Test;
63
64          public class «dm.fullyQualifiedName»Test {
65              «generateTests»
66          }
67      '''
68
69      // TODO: Gestionar Excepciones
70      def generateTests () {
71
72          var Graph graph = new Graph (dm);
73
74          var int count = Integer.parseInt(dm.count);
75          var String strategy = dm.strategy;
76
77          if (strategy.compareTo('state')==0) {
78              generateNodeCoverageTests(graph, count);
79          } else if (strategy.compareTo('transition')==0) {
80              generateEdgeCoverageTests(graph, count);
81          } else { //both
82              generateNodeCoverageTests(graph, count) + "" + generateEdgeCoverageTests(graph, count);
83          }
84      }
```

Figura 4.8. Parte 2 de la generación de código

En la figura 4.9. tenemos los métodos que llaman, a su vez, a los métodos que realizan las estrategias de recorrido de grafos. Estos últimos devuelven una lista de caminos. Dicha lista es recorrida, generando un método test junit por cada camino, al cual se designa con una cadena de texto fija concatenada con el valor del contador, del cual hemos hablado en un párrafo anterior, para diferenciar cada camino. Todo test comienza con la creación de un objeto. A continuación se recorre el camino, y por cada transición, se escribe la llamada al método y las comprobaciones asociadas a la transición.

Finalmente, en la figura 4.10., tenemos los métodos encargados del procesamiento de los estados iniciales, las transiciones y los métodos y comprobaciones asociados a las transiciones.

```
    def generateNodeCoverageTests(Graph graph, int loopCount) '''
        «graph.nodeCoverage(loopCount)»
        «iniCounter»
        «FOR l : graph.nodeCoverage»
            @Test
            public void nodePath«getCounter.toString»() {
                «compile (l.get(0).state1 as InitialState)»
                «FOR t : l»
                    «t.compile»
                «ENDFOR»
            }
            «incCounter»
        «ENDFOR»
    '''

    def generateEdgeCoverageTests(Graph graph, int loopCount) '''
        «graph.edgeCoverage(loopCount)»
        «iniCounter»
        «FOR l : graph.edgeCoverage»
            @Test
            public void edgePath«getCounter.toString»() {
                «compile (l.get(0).state1 as InitialState)»
                «FOR t : l»
                    «t.compile»
                «ENDFOR»
            }
            «incCounter»
        «ENDFOR»
    '''
```

Figura 4.9. Parte 3 de la generación de código

```
    def compile (InitialState s) '''
        «IF s.constructor==null»
            «dm.fullyQualifiedName» var«getCounter.toString» = new «dm.fullyQualifiedName»();
        «ELSE»
            «dm.fullyQualifiedName» var«getCounter.toString» = new «s.constructor»;
        «ENDIF»
    '''

    def compile (Transition t) '''
        «IF t.op != null»
            if (!(«t.guard1.compile» «t.op» «t.guard2.compile»)) {
                return;
            }
        «ENDIF»
        var«getCounter.toString».«t.command.compile»
        «FOR e:t.assert»
            «e.compile»
        «ENDFOR»
    '''

    def compile (Command m) '''
        «m.proc»;
    '''

    def compile (Assert a) '''
        «a.type» («a.assert1.compile» , «a.assert2.compile»);
    '''

    def compile (String s) {
        if (s.contains("(")) {
            return "var"+getCounter.toString+"."+s.toString;
        } else {
            return s.toString;
        }
    }
}
```

Figura 4.10. Parte 3 de la generación de código

5 Integración, pruebas y resultados

Ahora vamos a probar nuestro lenguaje de dominio específico con un caso real, con el objetivo de mostrar la dinámica y el funcionamiento del mismo. Dado que el objetivo del DSL es el de automatizar la creación de pruebas basadas en máquinas de estados, vamos a elegir una clase no implementada por nosotros para realizar las pruebas.

Hemos optado por utilizar la clase Stack (Pila, también llamada cola LIFO) de la API de java, para que las pruebas sean lo más objetivas posibles. Para que las pruebas tengan interés, hemos optado por la máquina de estados que se muestra en la figura 5.1. El estado "vacío" corresponde con la creación de una pila vacía. Los estados "uno", "dos" y "tres" corresponden, respectivamente, con un valor 1, 2 y 3 en lo alto de la pila. Los cambios de estado ocurren, naturalmente, al introducir un nuevo elemento de los tres disponibles en lo alto de la pila.

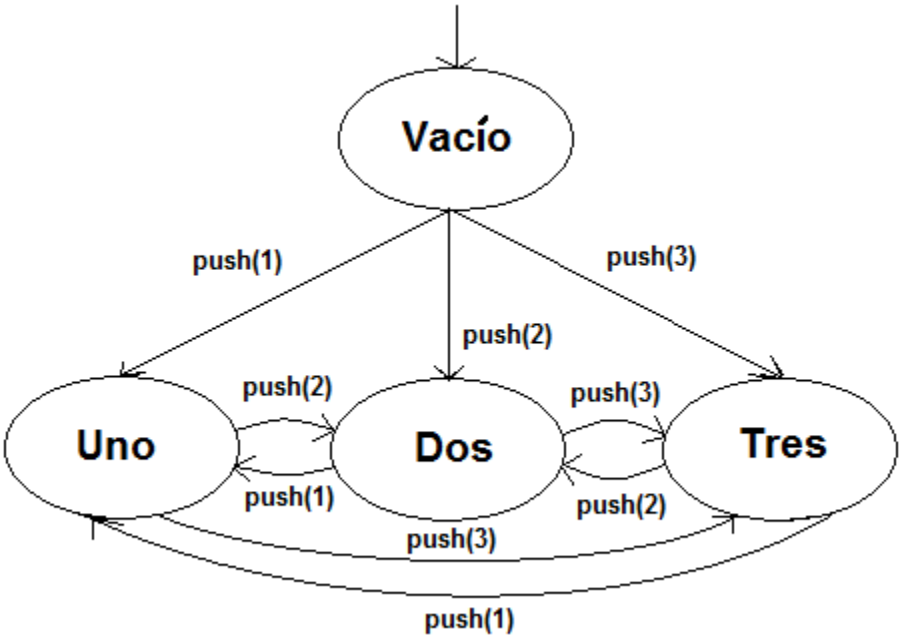

Figura 5.1. Estados de nuestra clase de prueba

En la figura 5.2. tenemos el código Mech que modela la máquina de estados. Hemos elegido las dos estrategias de cobertura y hemos establecido la estrategia de recorrido de bucles en tres iteraciones. Indicamos en el estado inicial que queremos crear una pila de enteros. A continuación declaramos el resto de estados y las transiciones. Hemos añadido una guarda en dos transiciones aleatorias para mostrar el funcionamiento. Estas guardas consisten en que se continuará con la prueba mientras la pila tenga menos de cuatro elementos.

testtfg.medsl | StackTest.java

```
1  package test
2
3  import java.util.Stack
4
5  spec for Stack coverage both loopCount 3 {
6      initial state vacio {
7          Stack<Integer>()
8      }
9      state uno
10     state dos
11     state tres
12
13     transition vacio - uno {
14         proc  push(1)
15         assertEquals peek(), 1
16     }
17     transition vacio - dos {
18         proc  push(2)
19         assertEquals peek(), 2
20     }
21     transition vacio - tres {
22         proc  push(3)
23         assertEquals peek(), 3
24     }
25     transition uno - dos guard size() < 4 {
26         proc  push(2)
27         assertEquals peek(), 2
28     }
29     transition uno - tres {
30         proc  push(3)
31         assertEquals peek(), 3
32     }
33     transition dos - uno {
34         proc  push(1)
35         assertEquals peek(), 1
36     }
37     transition dos - tres {
38         proc  push(3)
39         assertEquals peek(), 3
40     }
41     transition tres - uno {
42         proc  push(1)
43         assertEquals peek(), 1
44     }
45     transition tres - dos guard 4 > size() {
46         proc  push(2)
47         assertEquals peek(), 2
48     }
49 }
```

Figura 5.2. Código Mech de la prueba

Dado que el código Mech ha generado un código junit de 180 líneas, vamos a comentar un ejemplo de prueba de cobertura de nodos y otro de cobertura de aristas. La generación completa se encuentra en el Anexo C.

En la figura 5.3 tenemos una de las pruebas generadas para la cobertura de nodos. Tras crear la pila, se realizan las transiciones correspondientes y terminamos en un bucle entre los estados tres y uno. Dado que hemos establecido la estrategia de recorrido de bucles en tres iteraciones, se repite la iteración tres veces entre los estados tres y uno.

En la figura 5.4 tenemos una de las pruebas generadas para la cobertura de aristas. Como se observa, el comportamiento es similar, con la excepción de que este camino ha pasado por una de las transiciones con guarda.

```java
@Test
public void nodePath0() {
    Stack var0 = new Stack<Integer>();
    var0.push(3);
    assertEquals (var0.peek() , 3);
    var0.push(1);
    assertEquals (var0.peek() , 1);
    var0.push(3);
    assertEquals (var0.peek() , 3);
    var0.push(1);
    assertEquals (var0.peek() , 1);
    var0.push(3);
    assertEquals (var0.peek() , 3);
    var0.push(1);
    assertEquals (var0.peek() , 1);
    var0.push(3);
    assertEquals (var0.peek() , 3);
    var0.push(1);
    assertEquals (var0.peek() , 1);
    var0.push(3);
    assertEquals (var0.peek() , 3);
}
```

Figura 5.3. Una de las pruebas de la cobertura de nodos

```java
@Test
public void edgePath1() {
    Stack var1 = new Stack<Integer>();
    var1.push(3);
    assertEquals (var1.peek() , 3);
    if (!(4 > var1.size())) {
        return;
    }
    var1.push(2);
    assertEquals (var1.peek() , 2);
    var1.push(3);
    assertEquals (var1.peek() , 3);
    if (!(4 > var1.size())) {
        return;
    }
    var1.push(2);
    assertEquals (var1.peek() , 2);
    var1.push(3);
    assertEquals (var1.peek() , 3);
    if (!(4 > var1.size())) {
        return;
    }
    var1.push(2);
    assertEquals (var1.peek() , 2);
    var1.push(3);
    assertEquals (var1.peek() , 3);
    if (!(4 > var1.size())) {
        return;
    }
    var1.push(2);
    assertEquals (var1.peek() , 2);
    var1.push(3);
    assertEquals (var1.peek() , 3);
}
```

Figura 5.4. Una de las pruebas de la cobertura de aristas

En cuanto a las estrategias de cobertura, la propia naturaleza de las mismas, que se ha desarrollado en la sección de Desarrollo, y las pruebas que se han realizado durante el desarrollo con varios grafos nos aseguran que son correctas. De la misma forma que se han realizado pruebas con varios estados iniciales y varias comprobaciones por transición.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

A lo largo de esta memoria hemos hecho un desarrollo pormenorizado del lenguaje de dominio específico que hemos desarrollado. Hemos observado que el concepto de realizar pruebas basadas en máquinas de estados no es nuevo, sin embargo no es popular ni tenemos constancia de la existencia de herramientas que faciliten la tarea, de naturaleza repetitiva, lenta y propensa a errores.

Hemos tenido dos retos principales durante el desarrollo del lenguaje. Por un lado, la definición del mismo, aunque la fabulosa documentación de que dispone Xtext y Xtend ha hecho la tarea más fácil. Por otro, elegir como recorrer los grafos que representan las máquinas de estados, decidiendo el uso de la cobertura de nodos y la cobertura de aristas, como se ha desarrollado en el documento.

En definitiva, el trabajo realizado tiene como objetivo fomentar y facilitar las pruebas basadas en máquinas de estados ofreciendo un lenguaje textual fácil e intuitivo.

6.2 Trabajo futuro

Aunque el lenguaje es funcional, aún cabe espacio para mejora. Alguno de los aspectos a mejorar son:

- Validación: Es decir, la comprobación del código escrito Mech para detectar errores y ofrecer advertencias y soluciones. La validación es inexistente en nuestro lenguaje, sin embargo, es altamente recomendable mejorar este aspecto en favor de la experiencia del usuario.

- Generación automática de datos: Consiste en generar datos de manera automática para los métodos a partir de rangos de valores que indique el usuario. No se ha implementado en esta primera versión del lenguaje debido a su dificultad y complejidad.

Referencias

[1] Ammann, Paul, Jeff Offutt, and Wuzhi Xu. "Coverage criteria for state based specifications." Formal Methods and Testing, pp. 118-156, 2008

[2] Kim, Young Gon, et al. "Test cases generation from UML state diagrams." IEE Proceedings-Software 146.4, pp. 187-192, 1999

[3] Roger S. Pressman "Ingeniería del software: Un enfoque práctico 7º edición", pp. 12-14, McGraw-Hill Education, 2010

[4] Marco Brambilla, Jordi Cabot, Manuel Wimmer: "Model-Driven Software Engineering in Practice". Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers, 2012

[5] Sharul, M. D. "A Visual Formalism for Complex Systems." Science of Computer Programming vol. 8, 1987.

[6] "Template Expressions", http://www.eclipse.org/xtend/documentation/203_xtend_expressions.html#templates

[7] "State Machine Diagram" http://www.uml-diagrams.org/state-machine-diagrams.html

[8] "What is Junit?" http://junit.org/junit4/faq.html#overview_1

[9] "Xtext Introduction" http://www.eclipse.org/Xtext/documentation/index.html

[10] "Xtend Introduction" https://eclipse.org/xtend/documentation/index.html

[11] "Eclipse Modeling Framework (EMF)" https://eclipse.org/modeling/emf/

[12] "Junit Assertions" https://github.com/junit-team/junit4/wiki/Assertions

[13] "Class Assert" http://junit.sourceforge.net/javadoc/org/junit/Assert.html

[14] "Ecore Tools Overview" http://www.eclipse.org/ecoretools/overview.html

[15] Hyoung Seok Hong, Yong Rae Kwon and Sung Deok Cha, "Testing of object-oriented programs based on finite state machines," Proceedings 1995 Asia Pacific Software Engineering Conference, pp. 234-241, Brisbane, Qld., 1995,.

[16] Ipate F., Holcombe M. "Using State Diagrams to Generate Unit Tests for Object-Oriented Systems". In: Baumeister H., Marchesi M., Holcombe M. (eds) Extreme Programming and Agile Processes in Software Engineering. XP 2005. Lecture Notes in Computer Science, vol 3556. Springer, Berlin, Heidelberg, 2005

Glosario

DSL	Domain Specific Language (Lenguaje de Dominio Específico)
UML	Unified Modeling Language (Lenguaje de Modelado Unificado)
Marco de trabajo	Conjunto de herramientas, conceptos, prácticas y criterios que facilitan una tarea determinada.
Array	Anglicismo que designa un vector de cualquier dimensión
IDE	Integrated Development Enviroment (Entorno de Desarrollo Integrado)
LIFO	Last In, First Out (El último en entrar, es el primero en salir)
Assert	Comprobación (del inglés)

Anexos

A Sintaxis concreta textual completa

En este anexo se incluye la sintaxis textual de **Mech**

Domainmodel:
"package" <Import>
<Imports>*

"spec for" <ID> "coverage" ("state" | "transition" | "both") "loopCount" <INT> "{"
    <InitialState>+
    <State>*
    <Transition>*
"}"

Imports: "^"? ("a" .. "z" | "A" .. "Z" | "_") ("a" .. "z" | "A" .. "Z" | "_" | "0" .. "9")*

INT: ("0" .. "9")+

Imports: (<ID> ".")* <ID>

Import: "import" <Import>

InitialState:
"initial" "state" <ID> ("{"
    <Method>
"}")?

State: "state" <ID>

AbstractState: <State> | <InitialState>

Transition:
"transition" <ID> "-" <ID> "}"
    ("guard" (<Method> | <NUM>) ("==" | "!=" | ">" | ">=" | "<" | "<=")
            (<Method> | <NUM>))?
    <Command>
    <Assert>*
"}"

Command: "proc" Proc

Assert: ("assertEquals" | "assertSame" | "asssertNotSame")
        (<Proc> | <NUM>) "," (<Proc> | <NUM>)

Proc: (<ID> | <Method>) ("." (<ID> | <Method>))*

Method: <ID> "(" ((<Proc> | <NUM>) ("," (<Proc> | <NUM>))*)? ")"

B Pseudocódigo de algoritmos de recorrido de grafos

En este anexo se incluyen los algoritmos de recorrido de grafos

```
funcion coberturaDeNodos (int iteracionesBucle) {
    inicializar listaCaminos

    mientras listaNodosNoVisitados no este vacía {
        desactivar flagDeIteracción
        obtener nodo aleatorio de listaNodosNoVisitados
        eliminar nodo seleccionado de listaNodosNoVisitados
        obtener camino de nodos y de aristas desde un nodo inicial al nodo seleccionado
        eliminar los nodos del camino obtenido de listaNodosNoVisitados

        si del nodo seleccionado salen aristas {
            activar flagDeIteracción
            obtener arista aleatoria de dicho conjunto

            mientras camino no contenga nodo destino de la arista seleccionada y
            flagDeIteracion este activado{
                añadir nodo destino al camino de nodos
                eliminar nodo destino de listaNodosNoVisitados
                añadir arista al camino de aristas

                si del nodo seleccionado no salen aristas {
                    desactivar flagDeIteracción
                } de otro modo {
                    obtener arista aleatoria de dicho conjunto
                }
            }
        }

        si flagDeIteracción activada e iteracionesBucle > 0 {
            recorrer el bucle iteracionesBucle veces y añadir al camino de nodos y aristas
        }
        añadir camino a listaCaminos
    }
    devolver listaCaminos
}
```

```
funcion coberturaDeAristas (int iteracionesBucle) {
    inicializar listaCaminos

    mientras listaAristasNoVisitadas no este vacía {
        activar flagDeIteracción
        obtener arista aleatoria de listaAristasNoVisitadas
        eliminar arista seleccionada de listaAristasNoVisitadas
        obtener camino nodos y de aristas desde un nodo inicial a la arista seleccionada
        eliminar las aristas del camino obtenido de listaAristasNoVisitadas

        mientras camino no contenga nodo destino de la arista seleccionada y
        flagDeIteracion este activado {
            añadir nodo destino al camino de nodos

            si del nodo seleccionado no salen aristas {
                desactivar flagDeIteracción
            } de otro modo {
                obtener arista aleatoria de dicho conjunto
                eliminar arista de listaAristasNoVisitadas
                añadir arista al camino de aristas
            }
        }
        si flagDeIteracción activada e iteracionesBucle > 0 {
            recorrer el bucle iteracionesBucle veces y añadir al camino de nodos y aristas
        }
        añadir camino a listaCaminos
    }
    devolver listaCaminos
}
```

C Prueba generada completa

Test completo generado para la sección de pruebas.

```
@Test
public void nodePath0() {
    Stack var0 = new Stack<Integer>();
    var0.push(3);
    assertEquals (var0.peek() , 3);
    var0.push(1);
    assertEquals (var0.peek() , 1);
    var0.push(3);
    assertEquals (var0.peek() , 3);
    var0.push(1);
    assertEquals (var0.peek() , 1);
    var0.push(3);
    assertEquals (var0.peek() , 3);
    var0.push(1);
    assertEquals (var0.peek() , 1);
    var0.push(3);
    assertEquals (var0.peek() , 3);
    var0.push(1);
    assertEquals (var0.peek() , 1);
    var0.push(3);
    assertEquals (var0.peek() , 3);
}
```

```
@Test
public void edgePath1() {
        Stack var1 = new Stack<Integer>();
    var1.push(3);
    assertEquals (var1.peek() , 3);
    if (!(4 > var1.size())) {return;}
    var1.push(2);
    assertEquals (var1.peek() , 2);
    var1.push(3);
    assertEquals (var1.peek() , 3);
    if (!(4 > var1.size())) {return;}
    var1.push(2);
    assertEquals (var1.peek() , 2);
    var1.push(3);
    assertEquals (var1.peek() , 3);
    if (!(4 > var1.size())) {return;}
    var1.push(2);
    assertEquals (var1.peek() , 2);
    var1.push(3);
    assertEquals (var1.peek() , 3);
    if (!(4 > var1.size())) {return;}
    var1.push(2);
    assertEquals (var1.peek() , 2);
    var1.push(3);
    assertEquals (var1.peek() , 3);
}
```

```
@Test
public void nodePath1() {
        Stack var1 = new Stack<Integer>();
    var1.push(2);
    assertEquals (var1.peek() , 2);
    var1.push(3);
    assertEquals (var1.peek() , 3);
    var1.push(1);
    assertEquals (var1.peek() , 1);
    var1.push(3);
    assertEquals (var1.peek() , 3);
    var1.push(1);
    assertEquals (var1.peek() , 1);
    var1.push(3);
    assertEquals (var1.peek() , 3);
    var1.push(1);
    assertEquals (var1.peek() , 1);
    var1.push(3);
    assertEquals (var1.peek() , 3);
    var1.push(1);
    assertEquals (var1.peek() , 1);
    var1.push(3);
    assertEquals (var1.peek() , 3);
}
```

```
@Test
public void edgePath0() {
    Stack var0 = new Stack<Integer>();
    var0.push(2);
    assertEquals (var0.peek() , 2);
    var0.push(3);
    assertEquals (var0.peek() , 3);
    var0.push(1);
    assertEquals (var0.peek() , 1);
    var0.push(3);
    assertEquals (var0.peek() , 3);
    var0.push(1);
    assertEquals (var0.peek() , 1);
    var0.push(3);
    assertEquals (var0.peek() , 3);
    var0.push(1);
    assertEquals (var0.peek() , 1);
    var0.push(3);
    assertEquals (var0.peek() , 3);
    var0.push(1);
    assertEquals (var0.peek() , 1);
    var0.push(3);
    assertEquals (var0.peek() , 3);
}
```

```
@Test
public void edgePath2() {
    Stack var2 = new Stack<Integer>();
    var2.push(1);
    assertEquals (var2.peek() , 1);
    if (!(var2.size() < 4)) {return;}
    var2.push(2);
    assertEquals (var2.peek() , 2);
    var2.push(3);
    assertEquals (var2.peek() , 3);
    var2.push(1);
    assertEquals (var2.peek() , 1);
    if (!(var2.size() < 4)) {return;}
    var2.push(2);
    assertEquals (var2.peek() , 2);
    var2.push(3);
    assertEquals (var2.peek() , 3);
    var2.push(1);
    assertEquals (var2.peek() , 1);
    if (!(var2.size() < 4)) {return;}
    var2.push(2);
    assertEquals (var2.peek() , 2);
    var2.push(3);
    assertEquals (var2.peek() , 3);
    var2.push(1);
    assertEquals (var2.peek() , 1);
    if (!(var2.size() < 4)) {return;}
    var2.push(2);
    assertEquals (var2.peek() , 2);
    var2.push(3);
    assertEquals (var2.peek() , 3);
    var2.push(1);
    assertEquals (var2.peek() , 1);
}
```

```
@Test
public void edgePath3() {
    Stack var3 = new Stack<Integer>();
    var3.push(2);
    assertEquals (var3.peek() , 2);
    var3.push(1);
    assertEquals (var3.peek() , 1);
    var3.push(3);
    assertEquals (var3.peek() , 3);
    var3.push(1);
    assertEquals (var3.peek() , 1);
    var3.push(3);
    assertEquals (var3.peek() , 3);
    var3.push(1);
    assertEquals (var3.peek() , 1);
    var3.push(3);
    assertEquals (var3.peek() , 3);
    var3.push(1);
    assertEquals (var3.peek() , 1);
    var3.push(3);
    assertEquals (var3.peek() , 3);
    var3.push(1);
    assertEquals (var3.peek() , 1);
}
```